Redes de Computadores (RCOMP)

Theoretical-Practical (TP) Lesson 07

2017/2018

- Berkeley sockets API, C and Java.
- Basic functions/methods for TCP applications.
- TCP client and server.
- Asynchronous reception.

TCP connections

Unlike connectionless UDP, TCP is connection-oriented, this means, before sending and receiving actual data, a connection must be established between two applications.

Once the TCP connection is established, several benefits arise:

- data is sent through the connection in a continuous flow, as if writing to a file or a pipe, though a TCP connection is full-duplex.
- data delivery is guaranteed, TCP handles all transmission problems retransmitting any lost data when required.
- each byte of data is delivered in the same order it’s supplied.

Yet, the first step is establishing a TCP connection between two applications. For that purpose, each application must undertake a **different role**.

- One application will take the role of receiving a TCP connection request.
- The other application must then take the initiative of requesting to the first a TCP connection establishment.

TCP clients and servers

As part of the client-server model, the client takes the initiative on contacting the server. Thus, applying this model to TCP, results in being the TCP client application the one taking the initiative of requesting a TCP connection establishment with TCP server application.

A TCP client application starts by establishing a TCP connection with the TCP server application, then, usually (but not mandatorily), it writes a request to the server through the established TCP connection and afterwards reads the server reply.

A TCP server application is, therefore waiting for TCP connection requests from clients, when one arrives, accepts it and the connection is then established. Again, usually (but not mandatorily), after the connection is established the server will read a request from the TCP connection, processes it and then writes a reply back to the client.

The TCP client application side

The TCP client application's first step is establishing the TCP connection with the server, to do so, again, two details about the server are required:

- The IP node address where the server application is running, this is typically provided by the end user interacting with the client application.
- The local port number the server application is listening for TCP connection requests on. Usually the server local port number is fixed and hard coded in both the client and server applications.

In C language a TCP client application establishes a TCP connection with a TCP server application by calling the connect() function:

int connect(int socket, struct sockaddr *address, int address_len);

This function uses a SOCK_STREAM type socket to establish a TCP connection with the server identified by the caller-defined structure **address**. Prior to calling this function the socket must be opened but no binding to local port is required (port zero is assumed by connect), nevertheless, the **address** structure must contain the server's node IP address and the server application local port number. **This function returns -1 in the case of failure in the connection establishment**.

The TCP client application side

In Java language, a TCP client application establishes a TCP connection with a TCP server application by instantiating a Socket class object with the following constructor method:

public Socket(InetAddress address, int port) throws IOException

Where **address** is a **InetAddress** class object holding the server’s node IP address, and **port** is the local port number the server application is receiving TCP connections on. **This method will raise an exception if the connection establishment fails**.

Both the C function and the Java method provide real feedback about the application to application connection establishment success.

After success, the socket is connected to the server application.

The TCP server application side

TCP server applications are a bit more complex, this comes mainly from the fact they have to handle several clients connections. Unlike a UDP server that can focus its whole attention on one single request at a time, a TCP server will have in its hands several TCP connections (one for each connected client).

At any moment, without being able to predict when, a request may arrive from one client through one of the TCP connections, this is called an asynchronous receiving problem and will be addressed later in more detail.

In Java language the specific **ServerSocket** class is used to listen for TCP connection requests from clients, one constructor is:

public ServerSocket(int port) throws IOException

Where **port** is the local port number for listening TCP connection requests. This may raise an exception, typically because the local port number is already in use.

Notice however that client connections are not yet accepted, meanwhile, they will be in a pending state, they must be later explicitly accepted by the server application.

The TCP server application side

In C language, the server needs a SOCK_STREAM socket, and must bind it to a fixed local port number, additionally a buffer size for incoming connections requests (not yet accepted), must also be specified using the **listen()** function. Here is a sample code (no error checking here):

```
struct addrinfo  req, *list;

bzero((char *)&req,sizeof(req));
req.ai_family = AF_INET6;                    // allow both IPv4 and IPv6 clients
req.ai_socktype = SOCK_STREAM;               // TCP
req.ai_flags = AI_PASSIVE;                   // local address
getaddrinfo(NULL, "9999" , &req, &list);     // local port number 9999
sock=socket(list->ai_family, list->ai_socktype, list->ai_protocol);
bind(sock, (struct sockaddr *)list->ai_addr, list->ai_addrlen);
listen(sock, SOMAXCONN);
```

SOMAXCONN is a library constant defining the maximum acceptable value for the number of pending connections requests.

Accepting TCP client connections

Listening for TCP connection requests does not mean accepting them. A TCP connection is established only after being accepted by the server application. For that purpose, the TCP server application must call the **accept() function** in C or the **accept()** method, defined by the **ServerSocket** class, in Java.

Both share some fundamental properties, first, they are **blocking**, if there's no pending connection request to be accepted the process/thread blocks until one arrives, second, in the case of success they **create and return a new socket** connected to the client, this new socket represents **the server-side of the established TCP connection with the client**.

Java: **public Socket accept() throws IOException**

C: **int accept(int socket, struct sockaddr *address, int * address_len);**

In C the address structure will be used to place the client address (IP and port number), if NULL is passed, then the client address won't be stored.

After accepting a TCP connection, the first socket continues listening for other TCP connection requests, so the server should use it to accept the next incoming connection, but on the other hand, the server should also read client requests on already established connections.

TCP server - parallel processing

TCP servers always end up with several sockets (one for incoming connection requests and several others already connected to clients) the problem is events on these sockets are asynchronous, that is, the server isn’t able to predetermine in which of the sockets the next event will be.

Although other solutions are possible, the more widely used strategy is **creating a parallel subtask for each socket**, it can be a thread or a process. By doing so, each subtask can be waiting for an event in the corresponding socket without affecting (blocking) other subtasks.

In C language we have chosen to use processes, in Java language, threads.

Starting with C language, a process is created by calling the **fork()** function, exactly what fork() does is creating a duplicate of the current process, the original (usually called parent) and the duplicate (usually called child) are absolutely equal and in the same state, same data, same open descriptors and sockets.

Only one thing allows distinguishing the between the two: the return value of the fork() function.

Multi-process TCP server

The fork() function creates an exact copy of the current process:

int fork(void);

If successful, this function is called in one process and returns in two processes. To the original process (parent) fork() returns a non zero positive number (the child's PID) to the duplicate process (child) it returns zero. In case of failure fork() would return -1.

Basic sample of using fork() in implementing a multi-process TCP server:

```
(…)
bind(sock, (struct sockaddr *)list->ai_addr, list->ai_addrlen);
listen(sock, SOMAXCONN);
while(1) {
      cliSock=accept(sock, NULL, NULL); // wait for a new connection
      if(!fork()) {              // this is the child (fork() returned zero)
            close(sock);
            // process all client requests on cliSock
            close(cliSock);
            exit(0); }             // child process terminates
      close(cliSock);            // this is the parent
      }
```

Multi-thread TCP server

In Java a thread class can be defined by either declaring it implements the Runnable interface or by declaring it's Thread's subclass (extends the Thread class). The Thread class also implements the Runnable interface, either way, a run() method must be implemented by the class.

Example thread class declaration implementing the Runnable interface:

```
public class AttendClient implements Runnable {
      private Socket cliSock;
      public AttendClient(Socket s) { cliSock=s;}
      public void run() {            // thread execution starts here
            // process client requests on cliSock
            cliSock.close();
            }                 // thread execution ends here
}
```

In this example, the AttendClient() constructor is defined with the sole purpose of passing the socket to be stored within the object, for later use by the thread.

Multi-thread TCP server

Our declared thread class can then be instantiated, but that doesn’t start the thread execution, it just creates the object.

To actually start running the thread in parallel, the start() method must be called. This method creates a new thread and executes the run() method.

Nevertheless, the start() method is defined by the Thread class, so the exact steps to call it depend on the way the our thread class has been defined.

If our class was declared as extending the Thread class, then the start() method was inherited, therefore, it can be directly called through our class instance.

On the other hand, if our class merely implements the Runnable interface, then it has no start() method. In this case, an additional object must be created by instantiating the Thread class. One of the Thread’s constructors receives a Runnable interface as argument, so our object can be passed to it. Once the Thread object is created, it’s start() method can now be called to start running our object’s run() method in a separate thread.

Multi-thread TCP server

Now the main application thread that accepts incoming TCP connections can start a thread to handle each one:

```
public class TcpServer {
    public static void main(String args[]) {
      static ServerSocket sock = new ServerSocket(9999);
      static Socket nSock;
      while(true) {
                  nSock=sock.accept();     // wait for a new connection
            Thread cliConn = new Thread(new AttendClient(nSock));
            cliConn.start();          // start running the thread
            }
      }
}
```

For each accepted TCP connection a new thread is started to deal in exclusivity with that client's requests made through the connected socket, and sending back the replies. As part of the dialogue between the client application and the server thread, when the client has no more requests, it somehow requests the server to end the connection and the thread will be terminated.

Reading a writing through a connection

Dealing with data transfer through TCP connections is somewhat different from sending and receiving data carried inside UDP datagrams. At first glance, it's more straightforward because bytes can be directly read and written and TCP ensures they flow efficiently and reliably.

Yet, this byte flow presents some challenges to network applications.

UDP applications are required to be synchronized with datagram granularity only (each send must match one receive in the counterpart and vice-versa). The number of bytes transported by each datagram is not required to be known prior to the reception. After receiving the UDP datagram, the receiver learns its size.

For TCP applications, however, **a byte granularity synchronization is essential**, the number of bytes written in one side must exactly match the number of bytes being read on the counterpart. This requires a careful application protocol design ensuring applications always know exactly how many bytes they are supposed to read.

TCP – reading and writing in C

In C language the generic read() and write() functions can be directly used to receive bytes from and send bytes through a connected TCP socket:

int read(int socket, void *buf, int nbytes);

int write (int socket, void *buf, int nbytes);

Where **socket** is the connected TCP socket, **buf** a pointer to a place from where to get the bytes to be sent, or a place where to put the received bytes. And finally, **nbytes** is the number of bytes to be received or sent.

They return respectively the number of bytes received and the number of bytes sent, however, the read() function is blocking, if the requested number of bytes to read (nbytes) is not available (because the counterpart has not written them yet) it will block until they arrive. Both function return -1 in the case of error.

TCP – reading and writing in Java

In Java language, reading and writing cannot be directly performed on the socket, for reading the socket's InputStream must be used, for writing the socket's OutputStream must be used. Methods in Socket class are available to get these streams:

public OutputStream getOutputStream()   throws IOException

public InputStream getInputStream()   throws IOException

They return streams that can then be used to create the appropriate OutputStream and InputStream for specific purposes, for instance for raw byte reading and writing DataOutputStream and DataInputStream can be used. For example if **sock** is the connected socket they can be obtained by:

```
DataOutputStream sOut = new DataOutputStream(sock.getOutputStream());
DataInputStream sIn = new DataInputStream(sock.getInputStream());
```

The input stream (**sIN**) is used for reading, whereas the output stream (**sOut**) will be used for writing.

TCP – reading and writing in Java

Once the connected socket's InputStream and OutputStream are obtained, actual reading and writing may take place.

For an InputStream, among others, two read() methods are available:

```
int read()                              throws IOException
int read(byte[] b, int off,  int len)   throws IOException
```

The first version reads a single byte and returns its value, the second reads **len** bytes and places them on offset **off** of byte array **b**. This second version returns the total number of bytes read.

Also, for an OutputStream, among others, two write() methods are available:

```
void write(int b)                        throws IOException
void write(byte[] b, int off,  int len)  throws IOException
```

The first version writes a single byte **b**, the second writes **len** bytes from offset **off** of byte array **b**.

Asynchronous reception

Many network applications can operate with synchronous reception, they **stop at synchronization points and wait for the counterpart**. These synchronization points are calls to **blocking read/receive functions or methods**.

An application can use synchronous reception if, and only if, the exact sequence of events that will take place is known, and that is not always the case as we have seen on TCP server applications.

One possible solution is the one we have then used: **parallel processing** with threads or processes. In this solution we create a parallel task for each event, each task itself uses synchronous reception, but it does not interfere with the other tasks.

Still, other solutions are possible.

Polling: for asynchronous events in a set of sockets, we can **set a low read timeout in all sockets**. Then we can implement a polling cycle repeatedly trying the read/receive on each of them. In the case of failure we skip to the next socket, otherwise we process data received. This can be implemented both in Java and C.

The C language select() function

In C language, there is yet another option when a set of asynchronous events are possible in a set of sockets.

The **select()** function has the ability to monitor a set of sockets. In fact, any type of integer descriptors can be monitored, including opened files and pipes, and for instance 0 (stdin), 1 (stdout) and 2 (stderr).

int select(int nD, fd_set *rD, fd_set *wD, fd_set *eD, struct timeval *to);

Pointers to three sets of descriptors may be provided by the caller:

rD – set of descriptors to check if are ready for reading (data available).

wD – set of descriptors to check if are ready for writing.

eD – set of descriptors to check if and error has occurred.

Any of them can be NULL, meaning the caller does not want to check that feature for any descriptor. Usually, for the purpose of asynchronous reception, we will be using only **rD** (ready for reading – data available).

select() – timeout and return value

int select(int nD, fd_set *rD, fd_set *wD, fd_set *eD, struct timeval *to);

The select() function is blocking, it blocks until any monitored descriptor changes the status on the monitored feature. However, a timeout can be specified by the caller (**to**), in that case even without a status change, it returns after the specified period. The last argument can be NULL (no timeout) this means block forever until some status change.

When called, the select() function returns either -1 (an error occurred), 0 (timeout expired) or the number of descriptors with status change.

To actually know which descriptors' status has changed, an analysis of provided set of descriptors is required, **this function changes the values stored** in **rD**, **wD**, **eD** and **to** (successive calls to this function must always pre-initialize these values).

Internally select() uses a vector to store the descriptors, so in needs to know the size of that vector, that is the role of the first argument (**nD**), it must always be the biggest descriptor being monitored plus one. Maximum value for **nD** is defined in FD_SETSIZE, usually, 1024.

select() – handling descriptor sets

The **fd_set** data type is used to specify a set of descriptors. A pointer to this data type is used in arguments **rD**, **wD** and **eD**, they should be defined by the select() caller with the descriptors to be checked and they will be then modified by the select() function to contain only the descriptors with a status change.

There are some macros available to handle this data type:

```
void FD_ZERO(fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
```

FD_ZERO – empties the **set** descriptor set (removes all descriptors)

FD_SET – places the **fd** descriptor in the **set** descriptor set (adds to the set)

FD_CLR – removes the **fd** descriptor from the **set** descriptor set

FD_ISSET – checks if the **fd** descriptor belongs to the **set** descriptor set, return zero if it doesn't belong and one if it does.

select() – sample coding

Let’s imagine we have four UDP datagram sockets (SOCK_DGRAM) and we want to receive a datagram in whatever socket one datagram arrives first.

```
// the four datagram sockets are s1, s2, s3 and s4
// sockets are already bound and ready to receive
fd_set reading;                             // the descriptor set
int maxD;                                   // the greatest descriptor used

FD_ZERO(&reading);                          // empty the set
FD_SET(s1,&reading); maxD=s1;                     // add s1 and set maxD
FD_SET(s2,&reading); if(s2>maxD) maxD=s2;         // add s2 and update maxD
FD_SET(s3,&reading); if(s3>maxD) maxD=s3;         // add s3 and update maxD
FD_SET(s4,&reading); if(s4>maxD) maxD=s4;         // add s3 and update maxD
select(maxD+1, &reading, NULL, NULL, NULL);
```

We are checking just for reading and no timeout is specified, so select() will block until a datagram arrives at one of the sockets. Now, when select() returns we must check in which descriptors status has changed.

select() – sample coding (cont.)

After calling select(), only the descriptors whose status has changed will be in the descriptor set.

```
// after select return (assuming no error)
if(FD_ISSET(s1, &reading)) {
      recvfrom(s1, &buff, MAX_SIZE, (struct sockaddr *) &cli, &cliSize); }
if(FD_ISSET(s2, &reading)) {
      recvfrom(s2, &buff, MAX_SIZE, (struct sockaddr *) &cli, &cliSize); }
if(FD_ISSET(s3, &reading)) {
      recvfrom(s3, &buff, MAX_SIZE, (struct sockaddr *) &cli, &cliSize); }
if(FD_ISSET(s4, &reading)) {
      recvfrom(s4, &buff, MAX_SIZE, (struct sockaddr *) &cli, &cliSize); }
```

Notice that the else statement is not used because several datagrams may arrive at the same time to different sockets. Of course, there is no real purpose for this code beyond illustrating the use of select().